Software Conventions

for Streaming SIMD Extensions

Version 2.1

01/99

Order Number: 243873-002

Table of Contents

Revision History

Revision	Revision History	Date
2.1	FCS revision.	01/99

1 Introduction

For best performance, Streaming SIMD Extensions require their memory operands to be aligned to 16-byte (16B) boundaries. However, the existing software conventions for IA-32 (stdcall, cdecl, fastcall) as implemented in the Microsoft Visual C++ Compiler, or previous versions of the Intel® C/C++ Compiler, do not provide any mechanism for ensuring that certain local data and certain parameters be 16B-aligned. Furthermore, the addition of a new register file has opened up the possibility for passing some parameters in registers rather than in memory. The Intel C/C++ Compiler's support for Streaming SIMD Extensions and the new `__m128` datatype has therefore led to some extensions to these IA-32 software conventions:

- functions that use Streaming SIMD Extensions data provide a 16B-aligned stack frame
- `__m128` parameters are aligned, possibly creating "holes" (padding) in the argument block
- `__m128` parameters may be passed in registers

This document describes these new conventions as implemented by the current release of the Intel(R) C/C++ Compiler.

This document also describes the `__declspec(align())` extended attribute which can be used to request alignment of data.

2 Stack Alignment

This section explains how the Intel C/C++ Compiler supports aligning stack frames on 16-byte (16B) boundaries. Aligning a stack frame this strongly is necessary to support routines that use the `__m128` datatype for local variables, and which may spill xmm registers to the stack frame. This requirement is necessary to support the alignment requirements for memory references in the Streaming SIMD Extensions. In addition, this scheme is used to support improved alignment for `__m64` and `double` type data.

For variables allocated in the stack frame, the compiler cannot guarantee the base of the variable is aligned unless it also ensures that the stack frame itself is 16B-aligned. Previous IA-32 software conventions, such as those supported by the Microsoft Visual C++ Compiler and previous version of the Intel C/C++ Compiler, only ensure that individual stack frames are 4B-aligned. Therefore, a function called from, for example, a Microsoft compiled function can only assume that the frame pointer it used is 4B-aligned.

Earlier versions of the Intel C/C++ Compiler have attempted to provide 8B-aligned stack frames by dynamically adjusting the stack frame pointer in the prologue of main and preserving 8B-alignment of the functions it compiles. This technique is clearly limited in applicability: the main routine must be compiled by the Intel C/C++ Compiler, there may be no functions in the call-tree compiled by some other compiler (as might be the case for routines registered as callbacks), and no support is provided for proper alignment of parameters.

The solution to this problem is to have the function’s entry point assume only 4B alignment. If the function has a need for 8B or 16B alignment, then code will be inserted to dynamically align the stack appropriately, resulting in one of the stack frames shown in Figure 1.

As an optimization, an alternate entry point will be created that can be called when proper stack alignment is guaranteed by the caller. Through call graph analysis, calls to the normal (unaligned) entry point can be optimized into calls to the (alternate) aligned entry point when the stack can be proven to be properly aligned. Furthermore, a function alignment requirement attribute can be modified throughout the call-graph so as to

cause the least number of calls to unaligned entry points. As an example of this, suppose function F has only a stack alignment requirement of 4, but that it calls function G at many call-sites, and in a loop. If G's alignment requirement is 16, then by promoting F's alignment requirement to 16, and making all calls to G go to its aligned entry point, the compiler can minimize the number of times that control passes through the unaligned entry points.

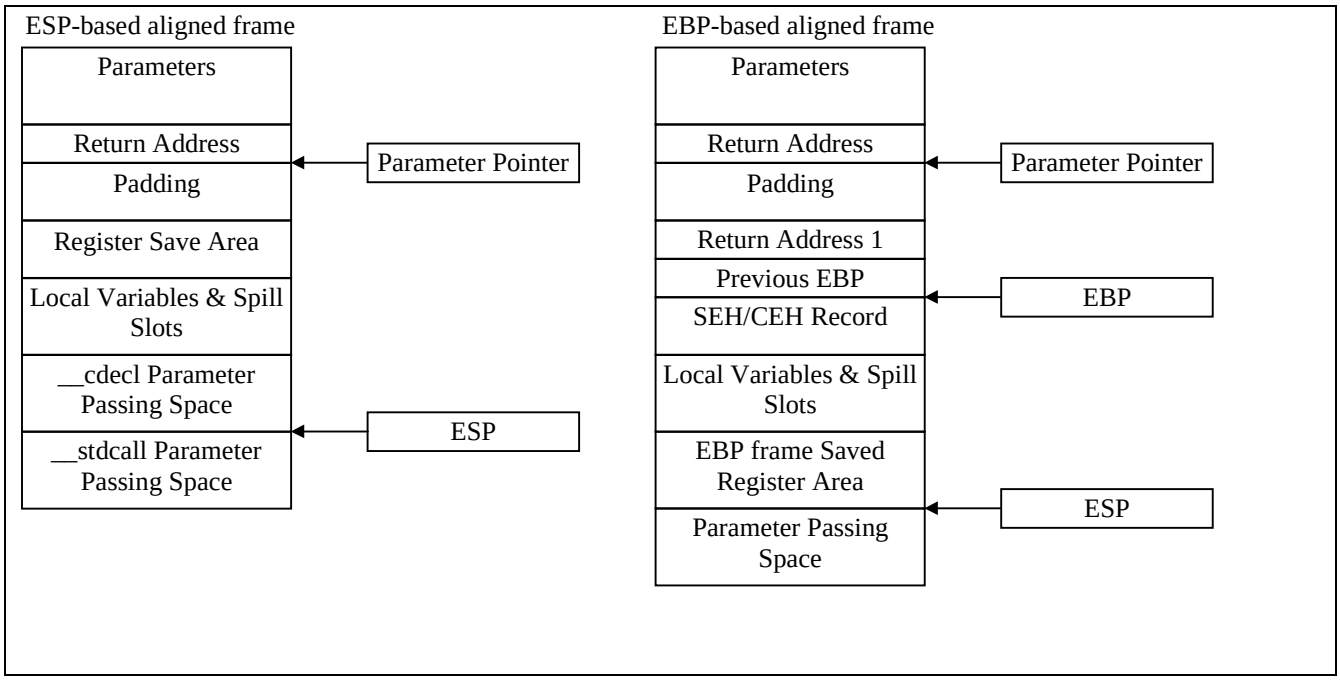

Figure 1.

2.1 Aligned esp-Based Stack Frames

In creating esp-based stack frames, the compiler adds padding between the return address and the register save area as shown in the example in Figure 2. This frame can only be used when debug information is not requested, there is no need for exception handling support, inlined assembly is not used, and there are no calls to `alloca` within the function. If these conditions are not met, an aligned ebp-based frame must be used. When using this type of frame, the sum of the sizes of the return address, saved registers, local variables, register spill slots, and parameter space must be a multiple of 16 bytes. This will cause the base of the parameter space to be 16-byte aligned. In addition, any space reserved for passing parameters for stdcall functions must also be a multiple of 16 bytes. This means that the caller will need to clean up some of the stack space when the size of the parameters pushed for a call to a stdcall function is not a multiple of 16. If the caller does not do this, the stack pointer will not be restored to its pre-call value.

In the above example, we have 12 bytes on the stack after the point of alignment from the caller: the return pointer, ebx and edx. Thus, we need to add 4 more to the stack pointer to achieve alignment. Assuming 16 bytes of stack space are needed for local variables, the compiler will add 16 + 4 = 20 bytes to esp, making esp aligned to a 0 mod 16 address.

Note A. An aligned entry point assumes that the beginning of the parameter block is aligned. This will place the stack pointer at a 12 mod 16 boundary, as the return pointer has been pushed. Thus, the unaligned entry point needs to force the stack pointer to this boundary.

Note B. The code at the common label assumes the stack is at an 8 mod 16 boundary, and adds sufficient space to the stack so that the stack pointer will be aligned to a 0 mod 16 boundary.

```
void _cdecl foo(int k) {
  int j;
foo:                                  // see note A
      push  ebx
      mov   ebx,esp
      sub   esp,0x00000008
      and   esp,0xfffffff0
      add   esp,0x00000008
      jmp   common
foo.aligned:
      push  ebx
      mov   ebx,esp
common:                               // see note B
      push  edx
      sub   esp,20

 j = k;
      mov   edx,[ebx+8]
      mov   [esp+16],edx

  foo(5);
      mov   [esp], 5
      call  foo.aligned


  return j;
      mov   eax,[esp+16]
      add   esp,20
      pop   edx
      mov   esp,ebx
      pop   ebx
      ret
}
```

Figure 2.

2.2 Aligned ebp-Based Stack Frames

In ebp-based frames, padding is also inserted immediately before the return address. However, this frame is slightly unusual in that the return address may actually reside in 2 different places in the stack. This occurs

whenever padding must be added and exception handling is in effect for the function. Figure 3 shows an example of the code generated for this type of frame. The stack location of the return address is aligned 12 mod 16. This means that the value of ebp will always satisfy the condition (ebp & 0x0f) == 0x08. In this case, the sum of the sizes of the return address, the previous ebp, the exception handling record, the local variables, and the spill area must be a multiple of 16 bytes. In addition, the parameter passing space must always be a multiple of 16 bytes. For a call to a stdcall function it will be necessary for the caller to reserve some stack space if the size of the parameter block being pushed is not a multiple of 16.

```
void _stdcall foo(int k) {
  int j;
foo:
      push  ebx
      mov   ebx,esp
      sub   esp,0x00000008
      and   esp,0xfffffff0
      add   esp,0x00000008    // esp is (8 mod 16) after add
      jmp   common
foo.aligned:
      push  ebx               // esp is (8 mod 16) after push
      mov   ebx,esp
common:
      push  ebp               // this slot will be used for duplicate return ptr
      push  ebp               // esp is (0 mod 16) after push (rtn,ebx,ebp,ebp)
      mov   ebp,[ebx+4]       // fetch return pointer
      mov   [esp+4],ebp       // and store relative to ebp    (rtn,ebx,rtn,ebp)
      mov   ebp,esp           // ebp is (0 mod 16)
      sub   esp,28            // esp is (4 mod 16)   // see note C
      push  edx               // esp is (0 mod 16) after push
                              // the goal is to make esp and ebp (0 mod 16) here
 j = k;
      mov   edx,[ebx+8]       // k is (0 mod 16) if caller aligned his stack
      mov   [ebp-16],edx      // J is (0 mod 16)

  foo(5);
      add   esp,-4            // normal call sequence to unaligned entry
      mov   [esp],5
      call  foo               // for stdcall, callee cleans up stack

  foo.aligned(5);
      add   esp,-16           // aligned entry, this should be a multiple of 16
      mov   [esp],5
      call  foo.aligned
      add   esp,12            // see note D

  return j;
      mov   eax,[ebp-16]
      pop   edx
      mov   esp,ebp
      pop   ebp
      mov   esp,ebx
      pop   ebx
      ret   4
}
```

Figure 3.

Note C. Here we allow for local variables. However, this value should be adjusted so that, after pushing the saved registers, esp is 0 mod 16.

Note D. Just prior to the call, esp is 0 mod 16. To maintain alignment, esp should be adjusted by 16. When a callee uses the stdcall calling sequence, the stack pointer is restored by the callee. The final addition of 12 compensates for the fact that only 4 bytes were passed, rather than 16, and thus the caller must account for the remaining adjustment.

2.3 Stack Frame Optimizations

The Intel C/C++ Compiler provides certain optimizations that may improve the way aligned frames are set up and used. These optimizations are as follows:

- If a procedure is guaranteed to leave the stack frame 16B-aligned and it calls another procedure that requires 16B alignment, then the callee's aligned entry point is called, bypassing all of the unnecessary aligning code.
- If a static function requires 16B alignment and it can be proven to be called only by other functions that require 16B alignment, then that function will not have any alignment code in it. That is, the compiler will not use ebx to point to the argument block and it will not have alternate entry points, because this function will never be entered with an unaligned frame.

2.4 Inlined Assembly and ebx

When using aligned frames the ebx register should generally not be modified in inlined assembly blocks since ebx is used to keep track of the argument block. Programmers may modify ebx only if they do not need to access the arguments and provided they save ebx and restore it before the end of the function (since esp is restored relative to ebx in the function's epilog).

3 Register Passing Conventions

The three common IA-32 calling conventions have been extended to support the new register set in the following ways:

- The first three `__m128` parameters are passed in registers xmm0, xmm1, and xmm2 ("args in regs"). Additional `__m128` parameters are passed on the stack as usual.
- `__m128` return values are passed in xmm0.
- Registers xmm0 through xmm7 are caller-save.

Space must be reserved by the caller in the argument block where the first three `__m128` parameters would normally appear. These locations are generally left empty by the caller, but may be used by the callee as "homes" for the xmm0, xmm1, and xmm2 registers if needed.

New versions of the stdarg.h and varargs.h headers are provided with the Intel C/C++ Compiler. These new implementations support variable argument lists containing `__m128` data, i.e. where padding may have been inserted as required for aligned parameters as described above. The new convention requires that functions with variable argument lists must be prototyped before calls are made to them and that, for this case only, the

caller must fill the locations on the stack for data in registers xmm0, xmm1, and xmm2. Callers to non-prototyped functions with variable argument lists with `__m128` data must pass parameters both on the stack and in registers.

4 Alignment for Variables

To force the alignment of a variable to be more strict than it otherwise would be, `__declspec(align())` can be used. This can be useful, for example, when an array of floats is also going to be used with the Streaming SIMD Extensions as if it contained `__m128` data. It can also be useful to force cache-line alignment.

The syntax for the extended attribute align is as follows:

```
__declspec(align(integer-constant))
```

where the *integer-constant* is an integral power of two no greater than 32.

The alignment of a variable can be increased, as in:

```
__declspec(align(16)) float buffer[400];
```

The address of the variable, `buffer`, is forced to be 0 mod 16. The variable buffer could then be used as if it contained 100 objects of type `__m128`. For convenience, the `xmmintrin.h` include file defines `_MM_ALIGN16` to be `__declspec(align(16))`.

Alternatively, a declaration such as the following could be used:

```
union {
        float f[400];
         __m128 m[100];
} buffer;
```

Because of the presence of `__m128` data in the union, 16B alignment is used by default; it is not necessary to use `__declspec(align())` to force it. Where feasible, for cases like this, it is generally better to use a union like the above rather than simply forcing alignment with `__declspec(align())`.

Sometimes it can be advantageous to force cache-line alignment of an arbitrary collection of data. For example, if there are three local variables named `i, j`, and `k`, they can be forced to reside in a single cache line by changing their declarations to something like:

```
__declspec(align(16)) struct {
        int i, j, k;
} x;
```

(Of course, references to these variables must be rewritten as `x.i, x.j`, and `x.k`, respectively.)

In C++ (but not in C) it is also possible to force the alignment of a class/struct/union type, as in:

```
struct __declspec(align(16)) my_m128 {
        float f[4];
};
```

Again, however, if the data in such a class is going to be used with the compiler intrinsics for Streaming SIMD Extensions, it is generally better to use a union to make such "type punning" explicit. In C++, an anonymous union can be used to make this usage more convenient:

```
class my_m128 {
```

```
        union {
                __m128 m;
                 float f[4];
        };
};
```

In this example, because the union is anonymous, the names `m` and `f` can be used as immediate member names of `my_m128`.

Note that `__declspec(align())` has no effect when applied to a class, struct, or union member in either C or C++.